Writing Pike modules in C

Marek Habersack <grendel@caudium.net>

2nd February 2003

Contents

1 Introduction

The Pike language can be enhanced by writing creating the extension modules. The modules can be written directly in Pike or in C (it is also possible to write modules in C++, but that's beyond he scope of this introductory material). The latter modules are compiled into shared binaries (if the target platform supports them) and then loaded by the Pike interpreter on demand. For the Pike user, the C modules appear to be the same as the ones written directly in Pike — there is no visible difference.

Writing modules in C requires the programmer to know about what does the Pike interpreter expect to find in the module, how to deal with the Pike calling conventions, how to modify and access variables, constants and functions defined in other places etc. This tutorial will introduce you to the basics of writing a Pike module in C.

This document describes Pike 7.4 and newer.

2 Prerequisities

Before you start writing your module, you have to make sure that your system contains a C compiler and that the Pike header files are installed anywhere on your filesystem. The Pike interpreter or libraries it depends upon are not required for module compilation. They are required, of course, for testing your module.

3 The basics

The Pike interpreter is an implementation of a special-purpose virtual machine (VM) and, as such, it defines an internal environment for defining variables, constants, classes, programs and functions (or methods). The environment features its own calling convention, stack, memory management, even a kind of an assembler (or pseudo code) which is an intermediate form of a Pike program. To extend the language with code written in C, one has to know at least two basic things — the calling convention (see page 13) and the stack organization (see page 13). For now it is sufficient for you to know that pike functions are called with parameters pushed to the stack and they return their results also on the stack.

4 Module structure

Pike C modules are similar in structure to libraries in that they don't contain the main() function and they contain a collection of functions and structures which can be used by 3rd party code. The Pike interpreter expects to find a couple of functions that are publically available and it enforces a certain convention on the functions which implement code visible from the Pike scripts. This section explains all the elements which are required in each C Pike module.

4.1 Header files

Each module must include several C header files in order to gain access to various Pike structures, macros and functions which are used throughout the code to implement interfaces required by the Pike interpreter. Below you can find a recommended list of the header files which should be included in the source of every C Pike module.

global.h global definitions, header file inclusions and platform-specific code.

bignum.h big numbers manipulation routines and macros.

array.h array manipulation routines.

builtin_functions.h declarations of functions built into the Pike interpreter that are visible from the Pike scripts. Some of those functions come useful even in the C modules.

constants.h constant manipulation routines and macros.

interpret.h functions and macros to manipulate the interpreter stack, the element reference counts, call other Pike functions from C.

mapping.h mapping manipulation routines and macros.

multiset.h multiset manipulation routines and macros.

module.h a few macros and function declarations for the required module elements.

module_support.h helper routines for C Pike modules.

object.h object manipulation routines and macros.

pike_macros.h various utility macros.

pike_types.h routines and macros related to the Pike typing system.

program.h program (a Pike type) manipulation routines and macros.

stralloc.h routines and macros to create, delete and manipulate the Pike *string* type.

svalue.h everything related to svalues - the bridge between C and Pike.

threads.h multi threading routines.

version.h Pike version.

operators.h operator functions declarations.

pike_error.h error reporting routines and macros.

With these header files included you gain access to all of the most common operations you might need to perform in your module.

4.2 Required functions

Pike requires only two public functions to be present in a shared library for it to be considered a Pike module. The functions are:

void pike_module_init(void) This function is called when the shared library is first loaded by the Pike interpreter. Its task is to define all the constants (5.1.2), variables (5.1.2), functions (5.1.3) and classes (5.1.6) that this module implements and make them visible to Pike scripts by registering them with the interpreter. The function is also responsible for allocating the class' (program's) private storage area (5.1.1).

void pike_module_exit(void) This function is called whenever the module is about to be unloaded. In practice it happens only when the Pike interpreter is exiting. You can perform any cleanup you need in this function.

For your convenience the `module.h` header file defines two macros that expand to the above declarations. They also account for the compilation in the Win32 environment which requires special declaration of the function's scope. The macros are called `PIKE_MODULE_INIT` and `PIKE_MODULE_EXIT` and are the preferred way of declaring those functions.

All other functions you define in your module can be declared as static.

5 Module implementation

Module implementation consists of two phases. First of them is creating and exporting the API functions and other elements so that they become visible to the Pike interpreter and to Pike scripts. The process in which this is done is called *registration*. In the following sections I will describe how to register various elements.

The second phase is adding the actual code to the functions exported in the first step. This is done in the usual way without many differences to the way you write a "normal" C program. The only differences lie in the way you affect the Pike interpreter - read parameters passed to your routine, manipulate the Pike variables, call Pike objects or functions and return results. To do that successfully, you have to follow a few guidelines which are also described in the following sections.

5.1 Exporting the module interfaces

In this section you will learn how to make the module elements visible from the Pike scripts — a process called *registration* of variables, constants, functions, programs (classes). Functions and other elements may either be visible in the *static* module namespace — that is they are accessible without the need of instantiating any class, or they might be visible in the object instantiated from a class defined in your module. Mechanisms to export the elements in any of those cases are the same and are described in the following sections.

5.1.1 Class storage

Every Pike program (class) may (and usualy has) a certain amount of memory set aside for its private use. The memory can also be called a *context* because that's where you store the C variables that describe the state of your module. The state is a broad term which describes anything your module might need throughout its lifespan to be able to perform its actions.

You must remember that your C code is, in a way, "injected" in the Pike script's namespace and exists in an evironment that the C compiler is unfamiliar with. The C compiler cannot directly store values in the Pike variables and in many cases there is no direct correspondence between certain C types (structures, pointers for example) and the Pike types. For these elements you need to have a place where, invisibly to the Pike script, your class (program) can store variables crucial to its operation. Another reason to have the storage is that very often the Pike script doesn't need to know (or even must not know sometimes) the details of the underlying C implementation — your module (often called a "glue" if it interfaces with some 3rd party library you want to access from your Pike scripts)

is a translation medium which exists in the middle between two different worlds.

To define class storage you first have to determine what data do you want to be accessible when your class becomes an object. Then you have to define a C structure that will contain all that data. For example:

```
typedef struct {
  int      my_status;
  double   average;
  time_t   last_modified;
} mymodule_storage;
```

After you have defined the storage structure, you must inform the Pike interpreter about its size, have it allocate that much memory and make it available to your program when it is instantiated. This is done by using the `ADD_STORAGE` macro. The short example below assumes the storage structure defined above is available at this point and that the `module.h` header file was included.

```
PIKE_MODULE_INIT
{
  ptrdiff_t   offset;

  start_new_program();
  offset = ADD_STORAGE(mymodule_storage);
  /* the remaining code */
}
```

The `start_new_program()` function is explained in section 5.1.4 below. Note that in the `ADD_STORAGE` macro invocation we use the storage **type** name and not a reference to any variable of that type. This is because the macro needs to learn the size of our structure and allocate the appropriate amount of memory.

One thing worth noting is that the storage is not allocated immediately but only at the time when your program is becoming an object — that is at the instantiation time. It means that you **must not** access the storage (5.2.3) in any way before the object is initialized (5.1.5).

The `offset` variable is set to the value returned by the storage registration macro and contains the offset to the storage area in the object definition structure used by the Pike interpreter. You need to store the value only if you plan to export C variables from your module to the Pike code.

5.1.2 Exporting variables and constants

There are times when you might want to make a C variable directly accessible from the Pike script or when you might want to create a Pike-only

variable (i.e. one not backed up by its C counterpart) from your C module. The `program.h` header file declares several functions that let you do both of the above things. The functions are described in the list below.

map_variable Using this function you can export a C variable stored in your class storage (the C structure) to make it accessible from a Pike script. It is not possible to share a C variable that is outside of the storage structure. The parameters to this function are described in the Appendix B.

define_variable This function defines a new Pike variable. The data for this function is not backed up by a C storage — the function exists only in the Pike program. The parameters to this function are described in the Appendix B.

simple_add_variable This routine allows you to add a new variable in a similar fashion to define_variable above, with the difference that it uses the C native types for the `name` and `type` attributes. For details, see Appendix B.

5.1.3 Exporting functions

To add a function to the Pike namespace you can use several functions and a few macros. I will list only four macros here since this is all you need in 99% of the cases:

ADD_FUNCTION Macro to make a specified function visible from Pike and set the C function that will handle the call. The function is added with the default optimization flags: `OPT_SIDE_EFFECT` and `OPT_EXTERNAL_DEPEND`. These flags mark the function as one which has side effects on the Pike environment (i.e. it modifies the environment in some way that may influence the interpretation of the other parts of the Pike program) and that depends on external influences (it references/uses external code). Details about the macro are in Appendix B on page 20.

ADD_PROTOTYPE Similar in usage to `ADD_FUNCTION` above but with the difference that it does not specify the handler C function. This macro is used when you need to introduce a function prototype (similar to the C function declaration) to the Pike program. The optimization flags are the same as above. Details about the macro are in Appendix B on page 20.

ADD_FUNCTION2 Macro to make a specified function visible from Pike and set the C function that will handle the call. The difference to

`ADD_FUNCTION` above is that this macro allows you to specify the optimization flags (see Appendix B on page 20). Details about the macro are in Appendix B on page 20.

ADD_PROTOTYPE2 Similar in usage to `ADD_FUNCTION2` above but with the difference that it does not specify the handler C function. This macro is used when you need to introduce a function prototype (similar to the C function declaration) to the Pike program. The optimization flags are the same as above. Details about the macro are in Appendix B on page 20.

5.1.4 Creating a program

The functions you export to the Pike interpreter from your module may either be visible in the module static namespace (static in this instance means that you don't have to instantiate any class to access the functions) or in a class (program) defined in that module. In the latter case you have to first instantiate the class (program) in order to create an object which will have the functions you have exported.

To create a program you need to use the `start_new_program` function and either of the `end_class` and `end_program` routines. If you choose to use the `end_program`, then you will have to use one additional function called `add_program_constant` in order to make the newly created program known to the Pike interpreter. The decision on which of the `end_*` functions to use depends on whether you will need to access the created program from your C code or not (see below for details). The program you export using `end_program` and `add_program_constant` must be freed by you when your module is being unloaded. Consider two short examples of how to use all of the above functions:

```
#include "global.h"
#include "machine.h"
#include "program.h"

static struct program *my_program = NULL;

PIKE_MODULE_INIT
{
  start_new_program();

  /* allocate storage and add functions etc. here */

  my_program = end_program();
  add_program_constant("MyProgram", my_program, 0);
```

```
}

PIKE_MODULE_EXIT
{
  if (my_program) {
    free_program(my_program);
    my_program = NULL;
  }
}
```

and an equivalent program but using `end_class` instead:

```
#include "global.h"
#include "machine.h"
#include "program.h"

PIKE_MODULE_INIT
{
  start_new_program();

  /* allocate storage and add functions etc. here */

  end_class("MyProgram", 0);
}

PIKE_MODULE_EXIT
{
}
```

The above examples are valid (albeit not very useful) Pike modules that should compile and link without problems.

The routines presented above are everything that is needed to make the program visible from a Pike script. Below you will find a short description of all the functions:

start_new_program Function to start the definition of a new program. Details about the function are in Appendix B on page 17.

free_program Function to free the previously created and initialized program. Returns a pointer to the newly allocated program. Details about the function are in Appendix B on page 17.

end_program Ends the definition of a program and returns a pointer to a structure describing the new program. Details about the function are in Appendix B on page 18.

end_class Similar in function to `end_program` above (it calls the latter) but does not return the pointer. Details about the function are in Appendix B on page 17.

5.1.5 Object initialization

Another pair of functions that may come useful when creating your own Pike classes in C are the class init/exit callbacks. The callbacks are invoked by the Pike interpreted whenever your class is instantiated and whenever its lifespan ends. The callbacks should be used whenever you need to be sure that your class storage is in sane state (at the object initialization) and that you leave no unreleased memory behind you (thus creating memory leaks). The two functions to install the callbacks are `set_init_callback` and `set_exit_callback`. Both of them **must** be called somewhere between the `start_new_program` and one of the `end_*` functions. The synopsis of both callback functions is as follows:

```
void my_callback(struct object*);
```

Again, both functions should be made static so that you don't pollute the global C Pike interpreter namespace. The following code excerpt may be added to the sample code given above:

```
/* put this code above PIKE_MODULE_INIT */
static void my_program_init(struct object *o)
{
   /* initialize your storage and other stuff here */
}

static void my_program_exit(struct object *o)
{
   /* free the memory and do whatever housekeeping you need here */
}

/* put this code between start_new_program and end_class somewhere */
set_init_callback(my_program_init);
set_exit_callback(my_program_exit);
```

A short description of the above routines follows:

set_init_callback Sets a function to be called on the object initialization. Details about the function are in Appendix B on page 18.

set_exit_callback Sets a function to be called on the object desctruction. Details about the function are in Appendix B on page 19.

5.1.6 Making the program visible to Pike

Making the program visible to Pike consists of either calling the `end_class` function with the class name (as described in 5.1.4) or using `end_program` followed by a call to the `add_program_constant` function as shown in the sample program on page 10. The description of the function follows:

add_program_constant Makes the program available to Pike scripts by creating a constant referring to the specified program in the Pike namespace. Details about the function are in Appendix B on page 18.

5.2 Implementing the module interfaces

Implementing the module interfaces

5.2.1 Pike calling convention

Pike calling convention

5.2.2 Pike C function declaration

Pike C function declaration

5.2.3 Accessing the object storage

5.2.4 Accessing the function arguments

Accessing the function arguments

5.2.5 Operations on the Pike stack

Operations on the Pike stack

5.2.6 Error reporting (exceptions)

Error reporting (exceptions)

5.2.7 Returning results to the caller

Returning to the caller

6 Dealing with the Pike types from C

Dealing with the Pike types from C

6.1 The svalue structure

The svalue structure

6.2 Operating on the simple types

Operating on the simple types

6.3 Operating on the complex types

Operating on the complex types

6.3.1 Arrays

Arrays

6.3.2 Mappings

Mappings

6.3.3 Multisets

Multisets

6.3.4 Objects

Objects

7 Debugging the module

Debugging the module

A A sample module

A sample module

B Useful C Pike functions and macros

map_variable [function]

```
int map\_variable(const char *name,
                  const char *type,
                  INT32 flags,
                  size\_t offset,
                  INT32 run\_time\_type);
```

Maps a Pike variable to a C variable stored in the class storage. The parameters:

- **name** — the name of the variable under which it should be visible from Pike.
- **type** — the Pike declaration of the variable. For example: `array(int)`.
- **flags** — one or more ID_ (see below) flags OR-ed together. Usually 0.
- **offset** — the offset of the variable relative to the beginning of your storage. To compute it correctly, you must have stored the value returned by the `ADD_STORAGE` macro and add to it the offset of the variable within your storage structure. The latter offset can be computed using the `OFFSETOF` (B) macro.
- **run_time_type** — one of the `PIKE_T_` constants defined in the `svalue.h` header file (B).

start_new_program [function]

```
void start_new_program();
```

Creates and initializes the Pike program structure.

free_program [function]

```
void free_program(struct program *p);
```

Frees the passed program and removes it from the Pike interpreter.

end_class [function]

```
int end_class(const char *name, INT32 flags);
```

Takes a class name and, optionally, flags and adds a program constant to the Pike interpreter. Calls `end_program` implicitly and returns 1 for success, 0 for failure. The parameters:

- **name** — the class name.
- **flags** — one of the `ID_` flags (B).

end_program [function]

```
struct program *end_program();
```

Closes the program definition and returns a pointer to a dynamically allocated program description structure. The returned pointer must be freed using the `free_program` (B) function.

add_program_constant [function]

```
int add_program_constant(const char *name,
                         struct program *p,
                         INT32 flags)
```

Takes a program name, a pointer to the structure describing it, optional flags and adds a constant referring to the specified program to the namespace of the Pike interpreter. The parameters:

- **name** — the program name.
- **p** — pointer to a structure describing the program being added. The pointer must come from the `end_program` (B) function.
- **flags** — one of the `ID_` flags (B).

set_init_callback [function]

```
void set_init_callback(void (*init_callback)(struct object *));
```

Sets a function to be called whenever an object is created from the program (class). The parameters:

- **init_callback** — the function to be called on object initialization. The function is passed a pointer to the object for which it is being called.

set_exit_callback [function]

```
void set_exit_callback(void (*exit_callback)(struct object *));
```

Sets a function to be called whenever an object of the given class is destructed. The parameters:

- **exit_callback** — the function to be called on object destruction. The function is passed a pointer to the object for which it is being called.

define_variable [function]

```
int define\_variable(struct pike\_string *name,
                    struct pike\_string *type,
                    INT32 flags);
```

Defines a new Pike variable. The variable is not associated with any C variable. The parameters:

- **name** — the name of the variable under which it should be visible from Pike.
- **type** — the Pike declaration of the variable. For example: `array(int)`.
- **flags** — one or more ID_ flags (see Appendix B page 20) OR-ed together. Usually 0.

simple_add_variable [function]

```
int simple\_add\_variable(char *name,
                         char *type,
                         INT32 flags);
```

THis function is a shortcut to `define_variable`. The difference is that it takes the native C types as its parameter and not the `struct pike_string` type taken by the latter routine. The parameters:

- **name** — the name of the variable under which it should be visible from Pike.
- **type** — the Pike declaration of the variable. For example: `array(int)`.
- **flags** — one or more ID_ flags (see Appendix B page 20) OR-ed together. Usually 0.

OFFSETOF [macro]

```
OFFSETOF(structure,variable);
```

This macro computes the offset of the indicated variable in the specified structure. For the first paramater you pass the **type** name, for the second the variable name within that structure.

ADD_FUNCTION [macro]

ADD_FUNCTION2 [macro]

ADD_PROTOTYPE [macro]

ADD_PROTOTYPE [macro]

ID_ flag macros [macro]

- **ID_STATIC** Symbol is not visible by indexing
- **ID_PRIVATE** Symbol is not visible by inherit
- **ID_NOMASK** Symbol may not be overloaded
- **ID_PUBLIC** Anti private
- **ID_PROTECTED** Not currently used at all
- **ID_INLINE** Same as local
- **ID_HIDDEN** Symbols that are private and inherited one step later
- **ID_INHERITED** Symbol is inherited
- **ID_OPTIONAL** Symbol is not required by the interface
- **ID_EXTERN** Symbol is defined later
- **ID_VARIANT** Function is overloaded by argument.
- **ID_ALIAS** Variable is an overloaded alias.

OPT_ flag macros [macro]

PIKE_T_ type macros [macro]

- **PIKE_T_ARRAY**
- **PIKE_T_MAPPING**

- **PIKE_T_MULTISET**
- **PIKE_T_OBJECT**
- **PIKE_T_FUNCTION**
- **PIKE_T_PROGRAM**
- **PIKE_T_STRING**
- **PIKE_T_TYPE**
- **PIKE_T_INT**
- **PIKE_T_FLOAT**

Index